PRÁCTICA 3
VIAJANTE DE COMERCIO

Algorítmica

Grado en Ingeniería Informática
Universidad de granada

ADRA SANCHEZ RUIZ
CRISTINA GARRIDO AMADOR
JUAN MANUEL CASTILLO NIEVAS
LUIS LIÑÁN VILLAFRANCA
MARIANA ORIHUELA CAZORLA GRUPO A2

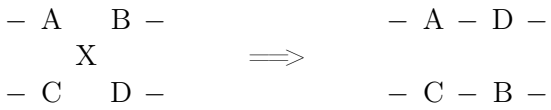
3 de mayo de 2017, Granada

Índice

1. Presentación del problema

En esta práctica hemos desarrollado varios algoritmos para resolver el problema del viajante de comercio. Se nos pedían dos algoritmos propuestos en el guión y uno más que encontráramos nosotros.

Nosotros hemos elegido como extra el algoritmo 2-opt, un algoritmo de búsqueda local definido por Croes en el año 1958 que escoge una ruta que se cruza con ella misma, y la reordena para evitar ese cruce. A continuación exponemos un ejemplo gráfico obtenido buscando información en internet:



2. Pseudocódigo del algoritmo del vecino más cercano

Candidatos: *ciudades a visitar*
Función solución: *se han visitado todas las ciudades*
Función selección: *se selecciona la ciudad más cercana a la última ciudad elegida*
Función de factibilidad: *siempre será factible nuestra solución*
Función de objetivo: *minimizar la distancia de la ruta entre ciudades (no se garantiza la optimalidad)*

Algorithm 1 Algoritmo Vecino más cercano

```
1: function HeuristicaVecinoMasCercano(mapa_ciudades, vector_ruta)
2:     Inicializa ciudad_elegida con el primer índice
3:     Inicializa ciudades_restantes con mapa_ciudades
4:
5:     for i ← 1 to size(mapa_ciudades) do
6:         delete(ciudades_restantes[ciudad_elegida])
7:         distancia_menor ← DistanciaEuclidea(mapa_ciudades[ciudad_elegida],
8:                                             ciudades_restantes[0].{x, y})
9:         ciudad_elegida_aux ← ciudades_restantes[0]{indice}
10:
11:        for each ciu in ciudades_restantes do
12:            distancia ← DistanciaEuclidea(mapa_ciudades[ciudad_elegida],
13:                                          ciu{x, y})
14:
15:            if distancia < distancia_menor then
16:                distancia_menor ← distancia
17:                ciudad_elegida_aux ← ciu{indice}
18:        ciudad_elegida ← ciudad_elegida_aux
19:        ruta.add(ciudad_elegida)
20:    return vector_ruta
```

3. Pseudocódigo del algoritmo de inserción

Candidatos: *ciudades a visitar*
Función solución: *se han visitado todas las ciudades*
Función selección: *la inserción más económica, es decir, la ciudad que provoque el menor incremento de la longitud total del recorrido*
Función de factibilidad: *siempre será factible nuestra solución*
Función de objetivo: *minimizar la distancia de la ruta entre ciudades (no se garantiza la optimalidad)*

Algorithm 2 Algoritmo Insercion

```
1: function HeuristicaInsercion(mapa_ciudades, vector_ruta)
2:     Inicializamos ciudades_restantes con mapa_ciudades
3:     Inicializamos lista_ruta con tres ciudades: la más a la izquierda,
4:                                                 la más a la derecha y
5:                                                 la más al norte
6:
7:     for i ← 0 to size(lista_ruta) do
8:         delete(ciudades_restantes[lista_ruta[i]])
9:
10:    while not empty(ciudades_restantes) do
11:
12:        for each it_ciudad in ciudades_restantes do
13:            Inicializamos mejor_ciudad con distancia infinita,
14:                                           begin(lista_ruta),
15:                                           begin(ciudades_restantes)
16:            tam_lista_ruta ← size(lista_ruta)
17:            it_lista_ruta ← begin(lista_ruta)
18:
19:            for i ← 0 to tam_lista_ruta do
20:                lista_ruta.Insert(it_lista_ruta, it_ciudad{indice})
21:                distancia_actual ← DistanciaCircuito(mapa_ciudades, lista_ruta)
22:
23:                if distancia_actual <mejor_ciudaddistancia then
24:                    mejor_ciudad.Set(distancia_actual, it_lista_ruta, it_ciudad)
25:                it_aux ← it_lista_ruta
26:                −− it_aux
27:                delete(lista_ruta[it_aux])
28:        lista_ruta.Insert(mejor_ciudad{posicion_lista}, mejor_ciudad{indice_mapa})
29:        delete(ciudades_restantes[mejor_ciudad{indice_mapa}])
```

4. Pseudocódigo del algoritmo de 2-opt

Candidatos: *ciudades a visitar*
Función solución: *no se cruza ningún recorrido*
Función selección: *escoge dos caminos que se cruzan y los reordena para que no ocurra esto*
Función de factibilidad: *siempre será factible nuestra solución*
Función de objetivo: *minimizar la distancia de la ruta entre ciudades (no se garantiza la optimalidad)*

Algorithm 3 Algoritmo 2-opt

```
1: function DosOptSwap(i, k, ruta)
2:     v_size ← size(vector_ruta)
3:
4:     for c ← 0 to i do
5:         nuevo[c] ← ruta[c]
6:     dec ← 0
7:
8:     for c ← i to k do
9:         nuevo[c] ← ruta[k - dec]
10:        ++dec
11:
12:    for c ← k + 1 to v_size do
13:        nuevo[c] ← ruta[c]
14:    return nuevo
15:
16: function HeuristicaDosOpt(mapa_ciudades, vector_ruta)
17:    size ← size(mapa_ciudades)
18:    mejora ← 0
19:
20:    while mejora < 20 do
21:        mejor_distancia ← INF
22:
23:        for i ← 0 to size − 1 do
24:
25:            for k ← i + 1 to size do
26:                nueva_ruta ← DosOptSwap(i, k, ruta)
27:                distancia_actual ← DistanciaCircuito(mapa_ciudades, ruta)
28:
29:                if distancia_actual < mejor_distancia then
30:                    mejora ← 0
31:                    ruta ← nueva_ruta
32:                    mejor_distancia ← distancia_actual
33:        ++mejora
```

5. Rutas dibujadas con Gnuplot

5.1. Ulysses16

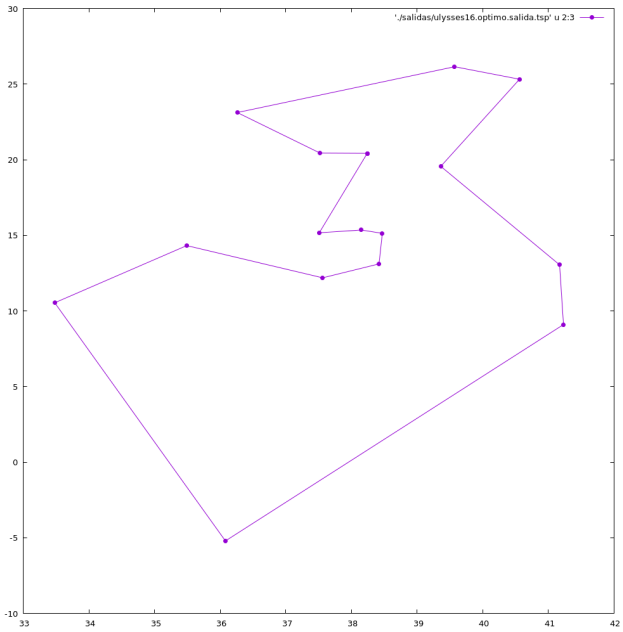


Figura 5.1: Resolución de ulysses16 con el algoritmo de Optimal

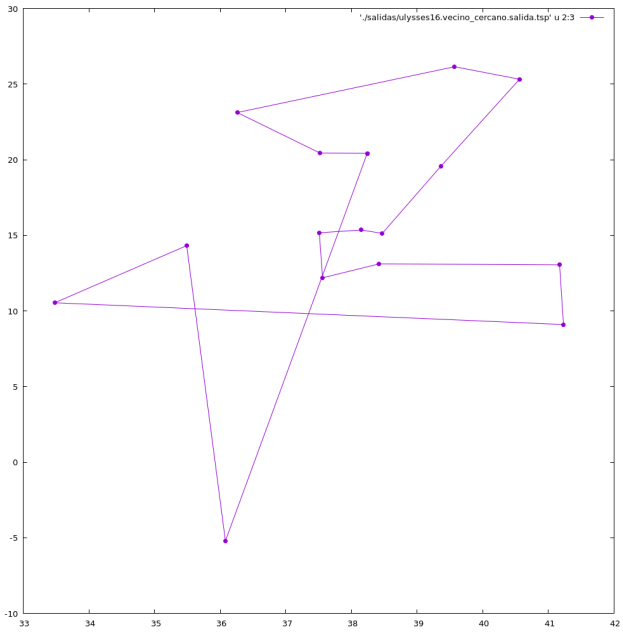


Figura 5.2: Resolución de ulysses16 con el algoritmo del Vecino más cercano

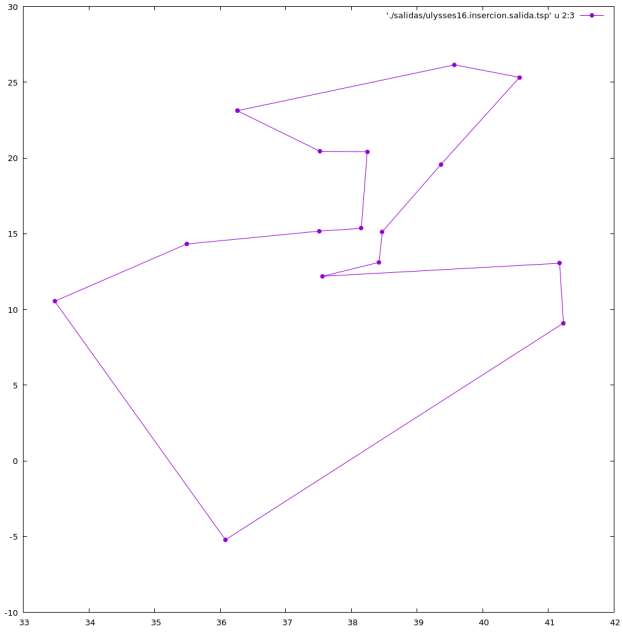


Figura 5.3: Resolución de ulysses16 con el algoritmo de Inserción

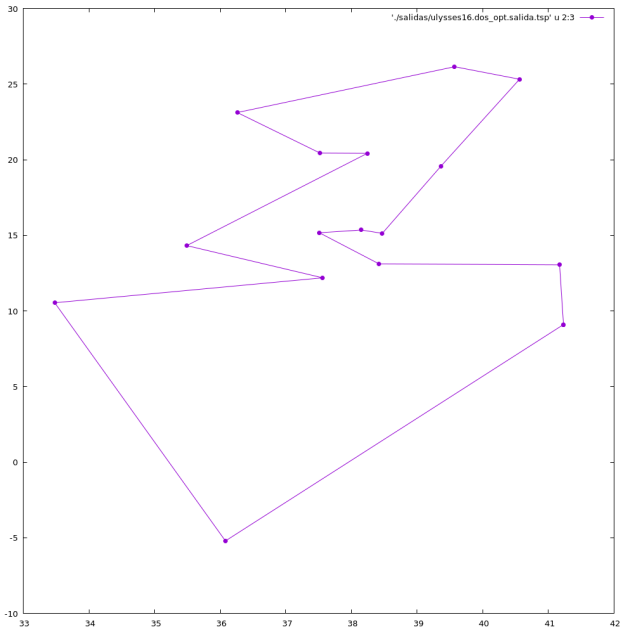


Figura 5.4: Resolución de ulysses16 con el algoritmo de 2-opt

5.2. Berlin52

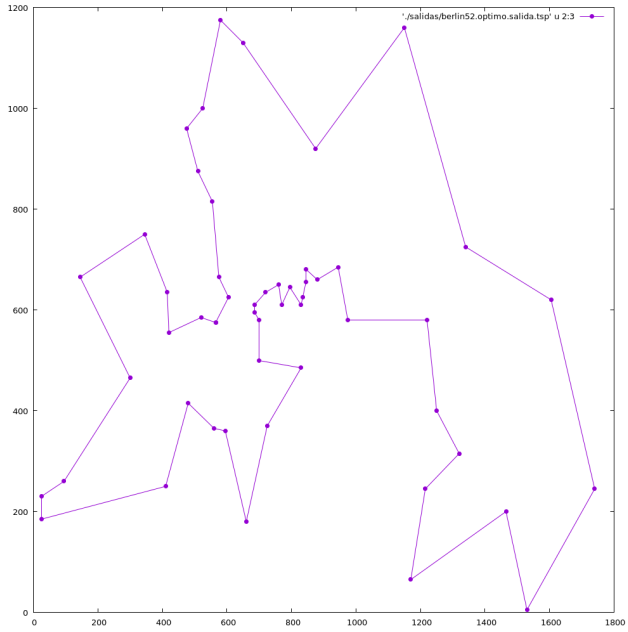


Figura 5.5: Resolución de berlin52 con el algoritmo de Optimal

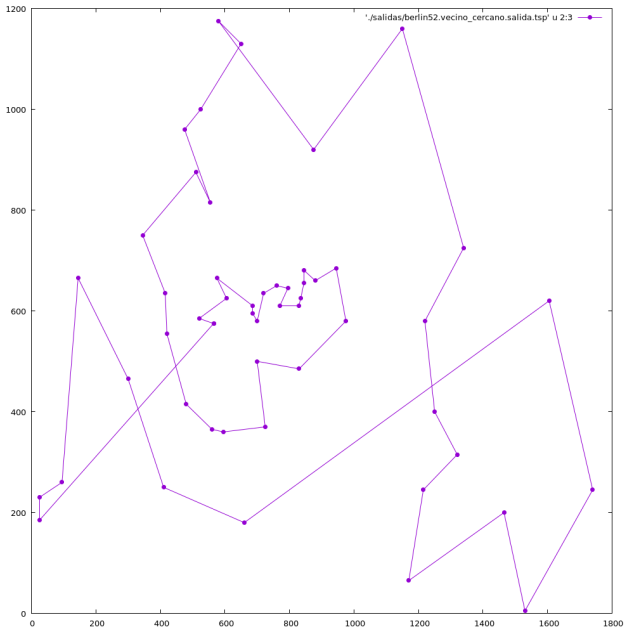


Figura 5.6: Resolución de berlin52 con el algoritmo del Vecino más cercano

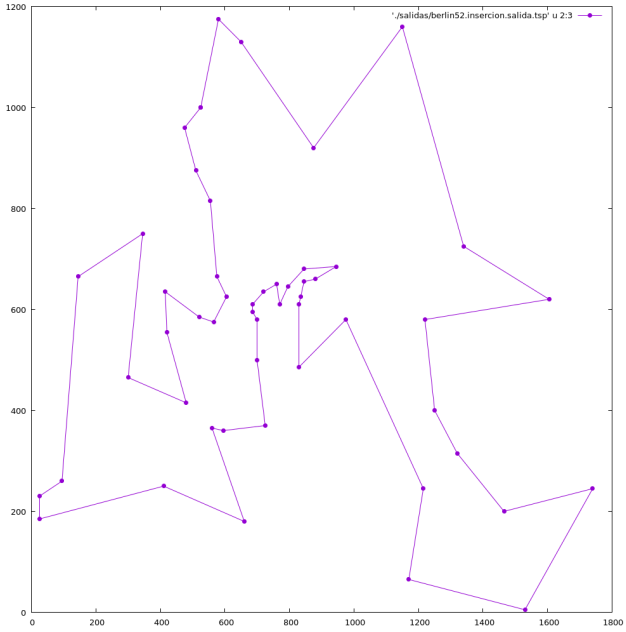


Figura 5.7: Resolución de berlin52 con el algoritmo de Inserción

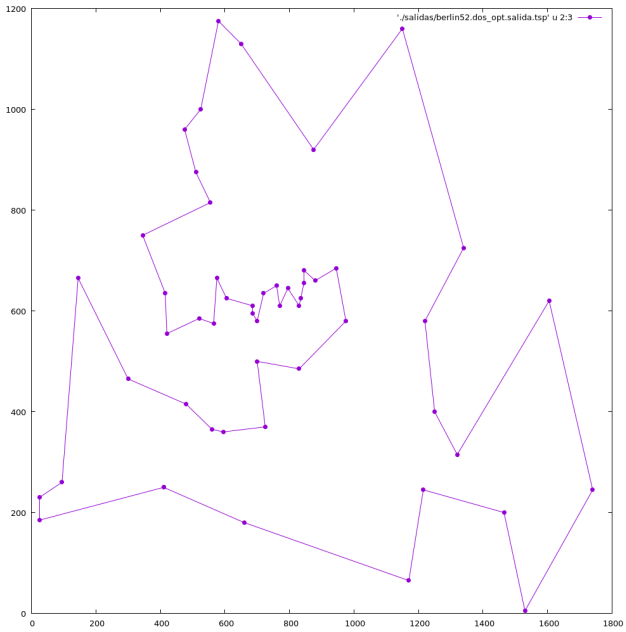


Figura 5.8: Resolución de berlin52 con el algoritmo de 2-opt

5.3. Eil101

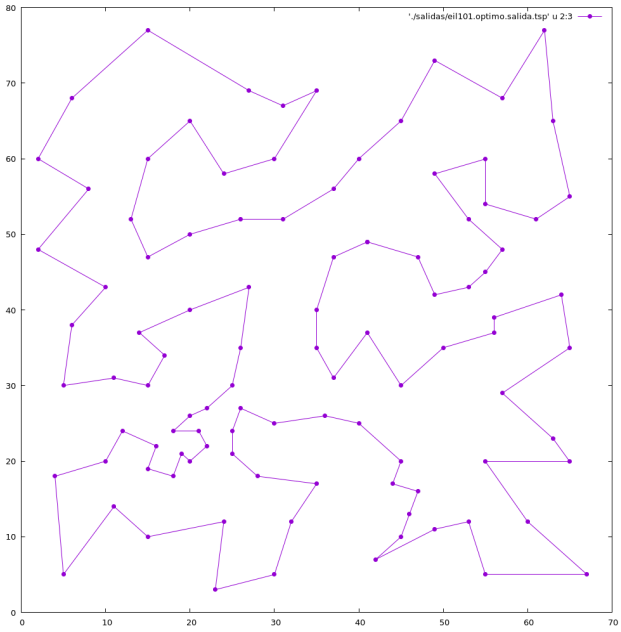


Figura 5.9: Resolución de eil101 con el algoritmo de Optimal

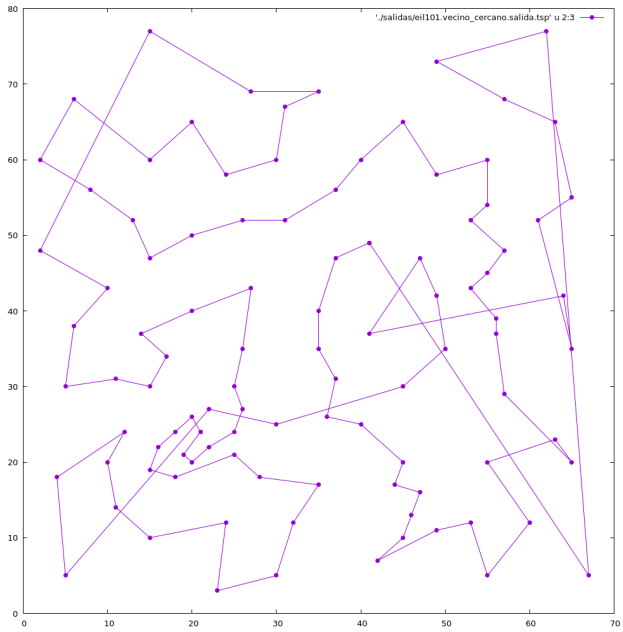


Figura 5.10: Resolución de eil101 con el algoritmo del Vecino más cercano

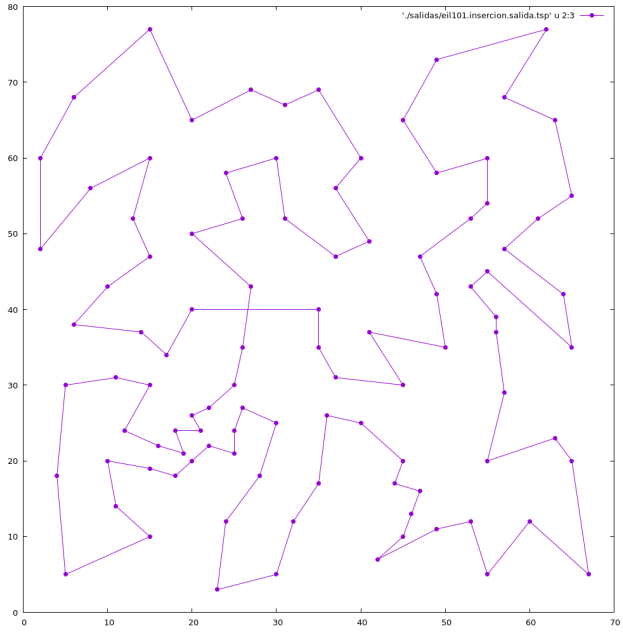


Figura 5.11: Resolución de eil101 con el algoritmo de Inserción

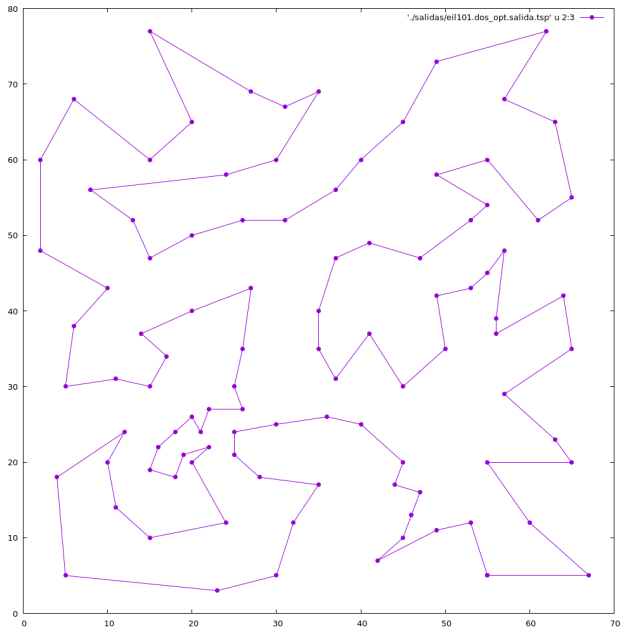


Figura 5.12: Resolución de eil101 con el algoritmo de 2-opt

5.4. Tsp225

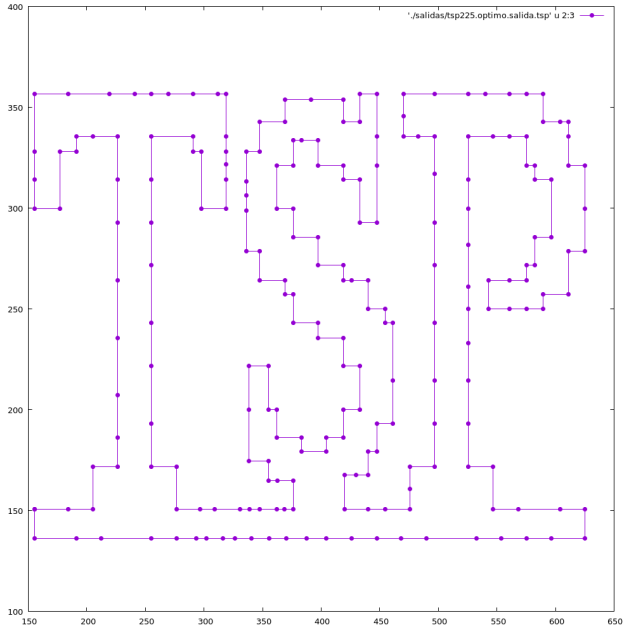


Figura 5.13: Resolución de tsp225 con el algoritmo de Optimal

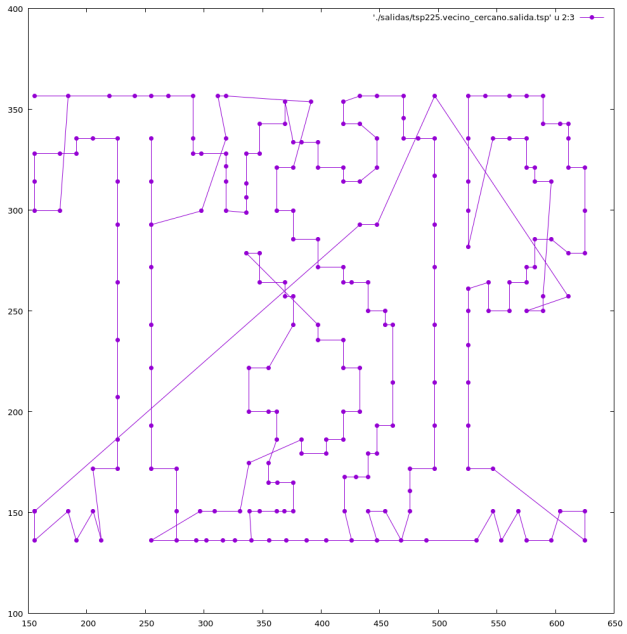


Figura 5.14: Resolución de tsp225 con el algoritmo del Vecino más cercano

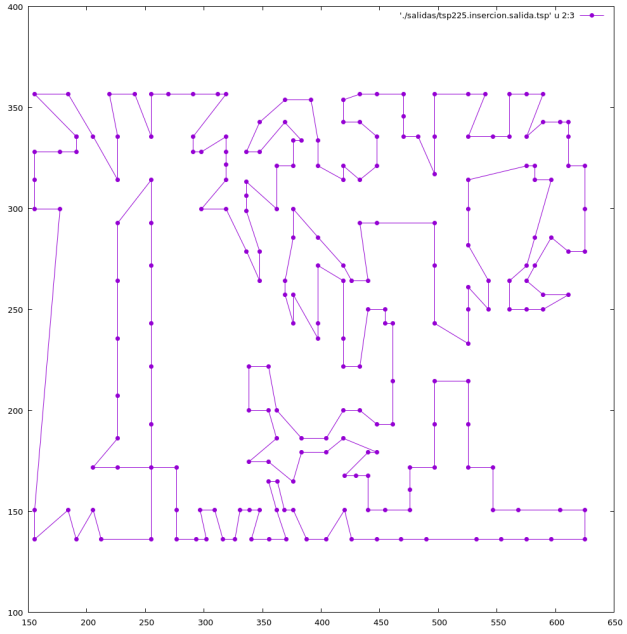


Figura 5.15: Resolución de tsp225 con el algoritmo de Inserción

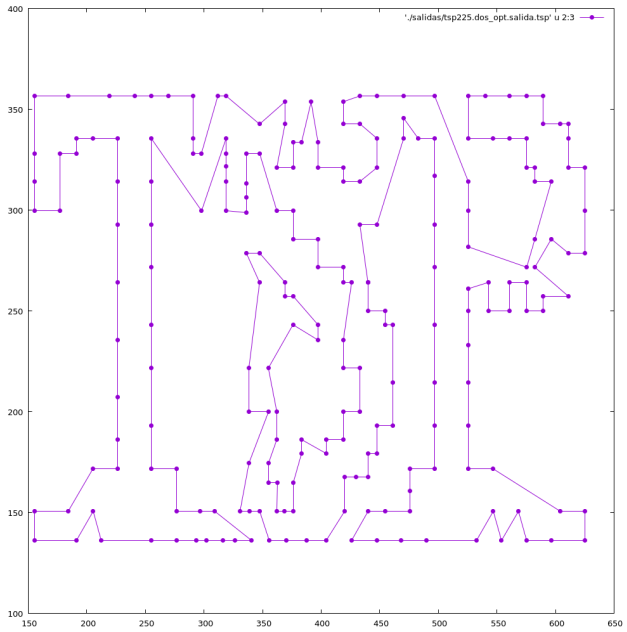


Figura 5.16: Resolución de tsp225 con el algoritmo de 2-opt